Detecting Anti-Patterns to Improve Continuous Integration

Thomas J. Dahlgren
dahlg136@morris.umn.edu
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

Abstract

Continuous Integration (CI) is a popular software engineering technique. It is where code changes from multiple developers are integrated into one central repository that runs "builds" and "tests" on the code. The advantage of this process is that differences in code and errors are found early in the development process and allow for faster releases of software. During this process there are multiple practices that can slowly remove the advantage of CI. These ill practices are called anti-patterns. These anti-patterns can be hard to spot by a team before they start to noticeably affect the project. Using tools to automatically detect these patterns for a team can help fix this problem. This paper discusses the use of two such tools to detect anti-patterns during the development process and in the configuration files of projects. These tools were able to successfully detect anti-patterns in open source CI projects and submit pull requests for those projects to get developer feedback. Developers responded in a positive way to having an overhead detection system for anti-patterns.

Keywords: anti-patterns, continuous integration, CI, software engineering

1 Introduction

Continuous integration (CI) is a software development process where developers integrate their code changes into a main code base that runs automated builds and tests. During this process there are many problematic practices a team of developers might follow that will take the benefits of using continuous integration away. These are called anti-patterns. There are tools being made now to detect anti-patterns automatically and alert teams to them so they can be fixed. One of the tools is CI-Odor and it detects anti-patterns in the process of merging branches, the skipping of failing tests, and the slowing of build times. Two other tools, Hansel and Gretel, detect anti-patterns in the configuration file for CI projects, and attempt to automatically remove them.

In this paper Section 2 will introduce the background needed to understand GitHub, Continuous Integration, and anti-patterns. Section 3 discusses the use of CI-Odor to detect anti-patterns in the software development process. A survey was sent out to developers to decide what anti-patterns they found prevalent in their projects and which ones should be detected. The paper then goes over the accuracy of CI-Odor and how developers responded to getting feedback about the anti-patterns in their projects. Section 4 will introduce the tools Hansel and Gretel which detect and remove anti-patterns in the configuration files of a CI project. It also covers the prevalence of these patterns in CI projects and how developers respond to them. The fifth section is a discussion of the two tool sets and their effectiveness with developers.

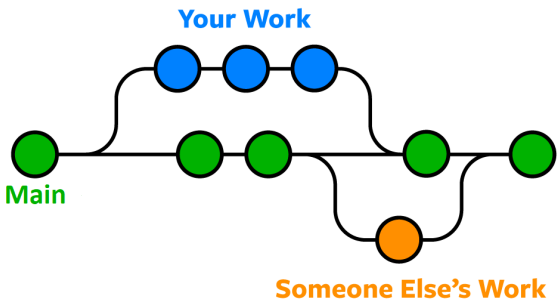

Figure 1. An example of merging branches into main. [1]

2 Background

2.1 GitHub

GitHub is an internet hosting service for software projects that uses Git for version control. Git allows for the creation of *branches* which allow a developer to duplicate part of the source code of a project so they can make changes to it without directly affecting the main code base. Uploading new code to GitHub is known as pushing a *commit*. A commit is when you save your current work so it can be pushed onto GitHub. Once the developer has finished writing their code they can push it to GitHub and make a *pull request* to *merge* it back into the main code (see Figure 1). A pull request is when you start the process of putting your new code into the main code base. Merging is the new code becoming the main source code after a pull request is accepted. Everything that is put on GitHub is saved so any changes can be reverted if the changes were to break the project or an older version of the project is deemed better than the current version.

```
before_install:
  - sudo apt-get update -qq

install:
  - wget https://bit.ly/hj67 -O /tmp/casper.tar.gz
  - tar -xvf /tmp/casper.tar.gz
  - bower install
```

Figure 2. Example of configuration file commands for the "before install" phase and the "install" phase. [2]

2.2 Continuous Integration

Continuous integration is the practice of integrating the code of multiple people into one main branch frequently (often multiple times a day). The other way of developing code would be to merge everyone's code into the main code branch when a team is ready to release their software. CI is used to improve the time it takes to integrate all of a team's code together, and issues the team has with each other's code can be found earlier in production. CI projects are usually hosted on services like GitHub. With CI, when code is pushed to a branch there ia a set of builds and tests that are automatically run to ensure the project compiles with these changes and still has the desired functionality, as determined by the tests. Common services that run these builds and tests would be GitHub Actions and Travis CI. Should the tests pass and the builds succeed then the developers of the project know the new code didn't "break" anything they tested for and can keep adding to the project. After every build a *build log* is made that has the results of how the build worked out and how all the tests did. The goal of CI is to commit often and merge branches into main frequently so that conflicts in the code can be found by the team early.

2.3 Configuration Files

A configuration file uses commands to determine what order the builds and tests should be run on a cloud platform (see Figure 2). In the configuration file there are *phases* that have certain commands that need to be run under them for example "install" commands go under the "install" phase. Other phases would be the "script" phase and the "deploy" phase. This file needs to be maintained by the team and changed to fit the project's needs as it evolves. There are features that the configuration file can use or misuse. For example it can specify branches that should be built when commits are made to that branch or it could have a command that introduces a security issue into the project.

2.4 Anti-Patterns in Continuous Integration

Anti-patterns are responses to a problem that are ineffective or counterproductive. Something is also an anti-pattern if there is a documented solution to a problem that proves the anti-pattern is not effective. Since anti-patterns may seem like they work or are effective it is hard for developers to know if anti-patterns are in their projects [4]. They can also show up in a project if a team is facing a deadline and needs to quickly get their software released or if upper management is putting pressure on the team to ignore certain CI principles.

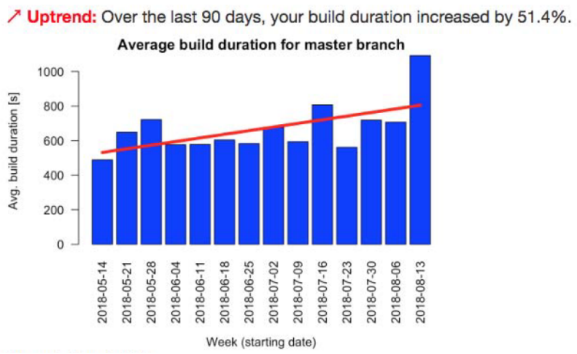

Figure 3. Example of slow build times. [3]

3 CI-Odor

CI-Odor is a tool that was created to test if the detection of anti-patterns was useful to developers who are currently using CI. It goes through the GitHub history and build logs of GitHub repositories and automatically detects anti-patterns that have entered a project. In order to decide what anti-patterns would be detected, a survey was publicly advertised in newsletters and public forums. The survey used Likert Scales "on a scale of 1 to 5 how strongly do you agree" and open questions to see what developers would find useful during their CI process. Four main anti-patterns came up from the survey: increasingly slow build times, skipped failed tests, late merging of branches into main, and broken release branches. [3]

3.1 Methods For Each Anti-Pattern

3.1.1 Increasingly Slow Build Times. Slow build times increase the amount of time it takes developers to get feedback on the build. This will slow down the project over time as developers wait for the build to run. To detect increasingly slow build times, the build times need to be monitored over time. Previous build logs will be in the history of the repository, so CI-Odor can be applied to projects that didn't use CI-Odor from their inception. The report of this anti-pattern is a bar graph that has a line through it (see Figure 3). A medium severity warning is given to builds that are slower than 75% of previous builds. A high severity warning is given to builds that are clear outliers of all the observed build times.

3.1.2 Broken Release Branches. When a release branch is broken it means the developers can't get feedback on the newly added features of that release. They have to put more time into fixing it that could have gone towards planning

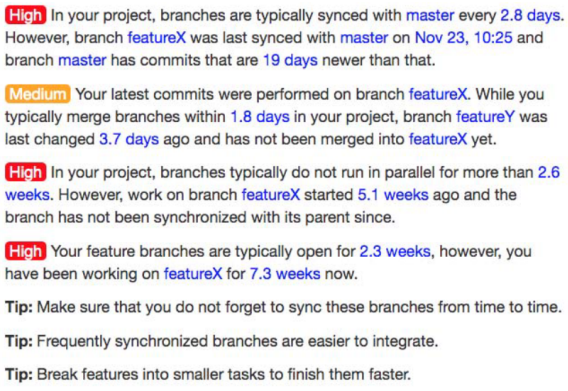

Figure 4. Example of a late branch merging report. [3]

the next set of features. The status of each build in the main branch is kept in the git history. This makes it easy for CI-Odor to find how many failed builds there have been and how long it took for each failed build to be repaired. CI-Odor uses this data to give a report to the team about trends in the build times. The report includes the average time it takes for the branch to be fixed, a bar chart of the amount of broken builds that happened each week, and a message about the trend in broken main branches (e.g., "The main branch was broken 90% less than usual over the last 90 days").

3.1.3 Skipped Failed Tests. Skipping a failing test will stop a build from failing, but it also decreases the security that the testing framework of CI provides. To detect how many failed tests are skipped CI-Odor uses this equation:

$$(\Delta Breaks < 0) \wedge (\Delta Runs < 0 \vee \Delta Skipped > 0)$$

Here $\Delta Breaks$ represents the change in build breaks, $\Delta Runs$ is the change in number of tests run, and $\Delta Skipped$ is how many skipped tests have been added to the build. If the Boolean test is true, then skipped tests have been added to the main branch of the project.

3.1.4 Late Merging of Branches. Branches that go a long time without being merged with main become increasingly hard to integrate into main. If the non-main branches deviate greatly from each other, then merging them all into main at the same time will also cause merge conflicts that slow down the development process. To detect branches that need to be merged with main it needs to detect: missed activity, branch deviation, branch activity, and branch age. Missed activity is how long a branch goes without the main branch being merged into it. Branch deviation is how different the branch is from other branches in the repository. Branch age keeps track of how long it's been since a branch was created. To determine a severity warning CI-Odor compares the values of the four metrics with the values that the other branches in the project have had over the course of the project (see Figure 4 for an example of a report on late branch merging).

3.2 CI-Odor Results

Once CI-Odor was configured to detect these patterns it was tested on public GitHub repositories. The first few projects were selected by the CI-Odor team so that they could guarantee feedback. A search for other projects that fit the criteria of using Java and Maven (for detecting skipped tests), using CI, being an active project, having build files, and having a team of at least five people was done to find other suitable candidates. Thirty-six projects were chosen that fit all the criteria. Of the 36 projects, 20 of them showed they had increasing build times. While we expect the build time to increase with the project size, some projects showed a worrisome amount of increase. 3% of the projects were flagged with a high-severity warning based on their increasingly slow build times. Broken release branches were detected in all of the projects. CI-Odor found the average time it took for the break to be fixed was around 1 day. Builds with skipped tests were rare among the tested projects. Only about 2.5% of builds were flagged with this anti-pattern showing developers don't normally skip failing tests between builds. All of the projects except one had late merging of branches into main with 115 high severity warnings being flagged across the 35 projects. [3]

3.3 Developer Feedback

After CI-Odor was run on the 36 projects a second survey was given to the developers who got to see CI-Odor feedback on their own projects. The second survey was composed of Likert scale questions in the same way as the first one, now asking about the usefulness of CI-Odor's generated reports. Around two-thirds of the developers who answered the questionnaire found the reports to be useful to the project. The ones who didn't like the reports didn't like how CI-Odor wasn't specific enough for their project or didn't like bugs that appeared in CI-Odor. A majority did agree that the detection of anti-patterns was useful since they brought awareness of the issue to the team.

Most of the developers said they found it easy to fix late merging and skipped failed tests, but increasing build times were harder for them to do anything about. Only about 25% of developers thought CI-Odor should have the ability to fail builds for having too many high severity anti-patterns. Most developers would rather it just report on issues in the project without doing anything else. 67.4% of the developers see CI-Odor leaving a positive effect on their project and 54.7% would integrate it into their projects immediately as it is.

4 Anti-patterns in project configuration files

The other approach to finding anti-patterns is to find them in the configuration files of a project. These anti-patterns are usually mistakes made with the configuration file for

Phase	Functionality	Command
Install	Install dependencies	`npm install`, `apt-get install`, `bower install`, `jspm`, `tsd`
Script	Testing	`npm test`, `mocha`, `jasmine-node`, `karma`, `selenium`
	Run Interpreter/Framework	`node`, `meteor`, `jekyll`, `cordova`, `ionic`
	Static Analysis	`codeclimate`, `istanbul`, `codecov`, `coveralls`, `jscover`, `audit-package`
Deploy	Deploying by script	`sh .*deploy.*.sh`

Figure 5. Examples of phases and the commands that go with them.

a CI project, and can be detected by the tool Hansel and removed by the tool Gretel. The anti-patterns that were decided for Hansel and Gretel to detect were redirecting scripts into interpreters, bypassing security checks, using irrelevant properties, and commands unrelated to the phase. Hansel uses YAML and BASHLEX parses to find anti-patterns in the configuration files of a project, specifically the .travis.yml files. Gretel then uses RUAMEL.YAML to remove the anti-patterns. [2]

4.1 Methods For Each Anti-Pattern

4.1.1 Redirection of Scripts into Interpreters. Redirecting scripts into the interpreter with a hard-coded URL is prone to security issues, and a network failure while downloading the script could result in a partially installed script. To detect the redirection of scripts into interpreters Hansel parses the configuration file to find commands that use pipes. If there is a command to download and then use a script it is an anti-pattern. To fix this anti-pattern the integrity of the script needs to be checked once downloaded or the script should be downloaded once and committed to the project. Since these solutions are outside of the scope of a .travis.yml file, Gretel cannot fix this anti-pattern. It has to be done by the team of the project.

4.1.2 Bypassing Security Checks. Configuration files can use SSH to connect to hosts remotely and download files or run code. Using it insecurely can lead to man-in-the-middle attacks. These are attacks where something pretends to be the host you're trying to connect to. To find a bypass of a security check anti-pattern Hansel looks for an `ssh_known_hosts` property in the add-ons section, the command `StrictHostKeyChecking = no`, or the command `User-KnownHostsFile = /dev/null`. These introduce security issues because they bypass the checks of ensuring the correct host is connected to. The only way to fix this anti-pattern is to have Gretel remove them from the configuration file when found. Then a known_hosts resource needs to be added to the project by the team so they aren't using any host that isn't known to the project. And finally the argument `-o UserKnownHostsFile = known_hosts` needs to be used whenever SSH is used in the configuration file.

4.1.3 Using Irrelevant Properties. Travis CI configuration files can contain properties that aren't supported by Travis CI. Users could have accidentally added these to their configuration files or have properties no longer supported by Travis CI left over in the project. Since Travis CI only gives a warning for these properties a developer might not see it, specifically if the build is successful, so detecting irrelevant properties can help bring awareness to the team. Irrelevant properties are parsed by Hansel and flagged as anti-patterns if they are in the wrong section of the configuration file or unrecognized. Gretel will remove properties that aren't mapped correctly and move their commands under the correct property. If a property is a slightly misspelled property for example, "before_srcipt", then Gretel will correct the spelling (before_script). If it doesn't recognize a property as a known one it will set a warning that the property is not used by Travis CI at run time.

4.1.4 Commands Unrelated to the Phase. Putting commands into an unrelated phase can cause maintenance problems. On top of that commonly used phases having different purposes than what is expected of them can make it hard for a developer to join the team since they won't be able to easily understand the behavior of that phase (see Figure 5 for common commands in phases). Travis CI also has optimizations to speed up phases that it can't use if a phase is not used correctly. Since different projects use different programming languages, Hansel needs to be configured differently to detect commands that weren't related to a phase for each framework. The researchers decided to configure it for Node.js since they found it to be the most popular language that uses Travis CI. For example Hansel would detect `npm install` commands outside of the install phase. Gretel will remove them from the phase they are not associated with and then add them to the end of the phase they should be in. If the phase is not defined in the configuration files then the phase will be added to the file.

4.2 Effectiveness of Hansel

Out of 9,312 tested projects 894 had at least one of the anti-patterns. To test the accuracy of Hansel the developers manually found anti-patterns in 100 of the projects that had anti-patterns and found that Hansel detected 82.76% of the anti-patterns in those projects. Specifically they found 29

anti-patterns in the 100 projects and Hansel detected 24 of them. 3 of the missed anti-patterns were the redirection of scripts into interpreters where the script was instead piped to an extractor after download. Since the script isn't executed it could be dropped from the requirements of Hansel giving it an accuracy of 93.10%. The 2 other misses were because Hansel wasn't configured to detect a command that belonged in the install phase for PHP code.

Since most of the instances of redirecting scripts into interpreters involved the METEOR web framework the researchers reached out to ask the framework developers about it. The METEOR developers said that the community is divided on the security implications of script redirection for their package and they don't have another solution for it. The researchers of Hansel suggest that if needed, downloading only over HTTPS, ensuring the script is downloaded fully, and regularly checking if the script still works after it is updated are ways to work with needing to redirect scripts.

The second anti-pattern of bypassing security checks was common. There were 63 detected cases in the 100 projects and 37 of them were setting `StrictHostKeyChecking=no`. 18 of them were putting `ssh_known_hosts` in the add-ons section of the configuration file to define host names and IP addresses used during the build process. This makes the project susceptible to DNS spoofing and other network attacks. The other 8 were the disabling of host key checking so it would be unknown if the correct host was connected to.

242 irrelevant properties were detected in the sample projects. Hansel was able to identify 74 misspelled properties. These properties are ignored by Travis CI when run. There were 148 properties that were misplaced. Some of these misplacements are ignored by Travis-CI just like the misspelled ones. 15 properties were flagged for being experimental properties. For example using the group property even though the Travis CI developers advised against using it at the time of the research since its usage might change before its official release. Five properties detected were using old Travis CI properties that may one day no longer be supported.

The final anti-pattern being detected, using commands outside of the relevant phase, detected 467 cases of install phase commands being used outside of the install phase. Since Travis CI optimizes the install phase the developers aren't getting the benefit of faster build times. There were 52 deploy phase commands found in the scripts phase. Since Travis CI optimizes the deploy phase for bandwidth and the script phase for CPU performance these benefits are lost to misplaced commands.

4.3 Effectiveness of Gretel

To test if Gretel could automatically remove anti-patterns, 250 anti-patterns were chosen to be tested. TravisLint was used on the configuration files before Gretel was run on them to make sure they were valid. Then the team manually checked the results of Gretel to make sure the scripts had the same behavior as before Gretel was run. Of the 250 anti-patterns, 174 could be removed automatically. 69 of the remaining cases needed to have manual verification to be removed. 38 of the 69 had state altering commands such as file system commands, package managers, and environmental variables. These commands in some cases were related to commands that Gretel has moved to a different phase of the configuration file and needed to be moved with them (see Figure 6 for an example of such a case).

```
language: node_js
node_js:
- '0.10'
before_script:
- cd frontend
- npm install -g
    bower grunt-cli
- npm install
- bower install
script:
- grunt test
```

```
language: node_js
node_js:
- '0.10'
install:
- cd frontend
- npm install -g bower
    grunt-cli
- npm install
- bower install
script:
- grunt test
```

Figure 6. Example of where an install command is in the before phase which is wrong, but can't be moved automatically because of a "cd" command. On the right is how the commands should be in the install phase.

12 were commands that were connected with a double ampersand (&&) meaning the command after the ampersands only runs if the one before was a success. Installation commands can be removed from these ampersands since if they fail then the build will fail and alert the team of the problem. And 29 of them had to do with the ruamel.yaml framework changing text as it moved it to the correct part of the configuration file. Errors such as turning the number ".10" into ".1" and line breaks adding a "\n" to the end of a moved command were mistakes that Gretel would make when moving text. The researchers were able to fix these errors that Gretel had but didn't test them again on further projects. Seven of the remaining 69 that couldn't be removed were because they used Yarn, a package manager to download scripts, and Gretel didn't have support for Yarn yet since its fixes would require a change they plan on adding in the future.

4.4 Developer's reaction

The 174 anti-patterns that could automatically be removed were submitted as pull requests. 49 of the pull requests got a response from the developers and 36 of them were accepted into the projects at the time of the paper's release. Two of the rejections were from pull requests that were made made on branches that had broken builds so new pull requests weren't being accepted. Another 2 were rejected since the developers didn't like how new commands were added to the project. The commands were added to the configuration file

to preserve the behavior the file had before moving the commands to the right phase. Two other requests were rejected since the projects were no longer maintained. One developer rejected a pull request because of Travis CI documentation that showed an install command in the before_script phase. The researchers reached out to Travis CI and they told the researchers that the documentation did need to be updated to move the commands into the install phase. The remaining 6 rejections didn't receive any feedback from the developers.

5 Discussion

Both the CI-Odor and Hansel and Gretel researchers saw positive results with detecting anti-patterns that hindered the CI process. The tools that they used were able to automatically identify anti-patterns accurately. Developers on real CI projects were surveyed and actual changes were suggested and accepted into the projects they work on. It was also learned that CI-Odor was found not only useful for detecting anti-patterns, but useful for teaching someone about the CI process. Both of the research teams agree that more projects should be sampled with their tools. They both only covered one programming language (CI-Odor was used on Java projects and Hansel and Gretel were used on Node.js projects) so a wider variety of languages should be examined. The anti-pattern of putting commands into the wrong phase that Hansel detected could lead to slower build times due to not being optimized on CI project building software. On the other hand increasingly slow build times was one of the anti-patterns that CI-Odor could detect, and the developers didn't know how to go about fixing it. So Hansel could help fix 2 anti-patterns in one if the misuse of commands is one of the causes of increasingly slow build times.

6 Conclusion

Continuous integration is a very useful tool for software engineering. It gives teams that use it many benefits for a fast development cycle. Anti-patterns take away some of these benefits. This paper has covered two categories of those anti-patterns: ones in the process of CI and ones in the configuration files. The first study with CI-Odor confirmed with developers that they would like certain anti-patterns in their development process to be detected. It also confirmed that they could be detected and that all projects they tested with it had at least one anti-pattern in them. The second study looked at the anti-patterns in the configuration files of CI projects. It showed that around 9.6% of projects have anti-patterns in their configuration files. These anti-patterns decrease the security of the projects and can slow down the CI process. The researchers were also able to confirm that the anti-patterns could be automatically removed making it easy for teams to fix.

The studies in this paper were initial looks into the process of detecting anti-patterns. Future work towards detecting more anti-patterns and detecting them with more techniques than just the build logs and configuration files could further help teams with CI. More programming languages will also need to be supported so more developers can benefit from the use of anti-pattern detection tools. Making tools that can adapt to specific project workflows will allow teams to integrate the tools into their projects without having to change how they choose to practice CI.

Acknowledgments

I would like to thank my advisor Kristin Lamberty and instructor Elena Machkasova for helping provide feedback and support on my paper. I would also like to thank John Hoff for providing external feedback on my paper.

References

[1] 2021. Git Branches: List, Create, Switch to, Merge, Push, Delete. https://www.nobledesktop.com/learn/git/git-branches [Online; accessed May 25, 2022].

[2] Keheliya Gallaba and Shane McIntosh. 2020. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2020), 33–50. https://doi.org/10.1109/TSE.2018.2838131

[3] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. 2019. Automated Reporting of Anti-Patterns and Decay in Continuous Integration. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 105–115. https://doi.org/10.1109/ICSE.2019.00028

[4] Wikipedia. 2022. Anti-pattern — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Anti-pattern&oldid=1078641245. [Online; accessed 25-March-2022].